

Efficient Construction of Simultaneous Deterministic Finite Automata on Multicores Using Rabin Fingerprints

Minyoung Jung and Bernd Burgstaller
Department of Computer Science
Yonsei University
Seoul, Korea

Johann Blieberger
Institute of Computer Aided Automation
Vienna University of Technology
Vienna, Austria

Abstract—The ubiquity of multicore architectures in modern computing devices requires the adaptation of algorithms to utilize parallel resources. String pattern matching based on finite automata (FAs) is a method that has gained widespread use across many areas in computer science. However, the structural properties of the pattern matching algorithm have hampered its parallelization. To overcome the dependency-constraint between subsequent matching steps and exploit parallelism, simultaneous deterministic finite automata (SFAs) have been recently introduced. Given an FA A with n states, the corresponding SFA $S(A)$ simulates n parallel instances of FA A . However, although an SFA facilitates parallel FA matching, SFA construction itself is limited by the exponential state-growth problem, which may result in $O(n^n)$ SFA states for an FA of size n . The substantial space requirements incur proportional processing steps, both of which make sequential SFA construction intractable for all but the smallest problem sizes.

In this paper, we propose several optimizations for the SFA construction algorithm, which greatly reduce the in-memory footprint and the processing steps required to construct an SFA. We introduce fingerprints as a space- and time-efficient way to represent SFA states. To compute fingerprints, we apply the Barrett reduction algorithm and accelerate it using recent additions to the x86 instruction set architecture. We exploit fingerprints to introduce hashing for further optimizations. Our parallel SFA construction algorithm is nonblocking and utilizes instruction-level, data-level, and task-level parallelism of coarse-, medium- and fine-grained granularity. We adapt static workload distributions and align the SFA data-structures with the constraints of multicore memory hierarchies, to increase the locality of memory accesses and facilitate HW prefetching.

We conduct experiments on the PROSITE protein database for FAs of up to 702 FA states to evaluate performance and effectiveness of our proposed optimizations. Evaluations have been conducted on a 4 CPU (64 cores) AMD Opteron 6378 system and a 2 CPU (28 cores, 2 hyperthreads per core) Intel Xeon E5-2697 v3 system. The observed speedups over the sequential baseline algorithm are up to 118541x on the AMD system and 2113968x on the Intel system.

Keywords—SFA construction; fingerprints; hashing; parallelization; multicores

I. INTRODUCTION

Multicore architectures have become main-stream. The ubiquity of SIMD units, multiple cores and GPUs in today's

desktops, servers and even handheld devices necessitates the adaptation of standard algorithms to expose parallelism and utilize parallel execution units. Parallelization improves performance and makes algorithms scalable to larger problem sizes.

Searching a pattern from a large text has been a prevalent processing step for various applications in computer science. Examples include text editors, compiler front-ends, scripting languages, web browsers, internet search engines, and security and DNA sequence analysis. FAs derived from regular expressions enable such uses, but the underlying, sequential FA algorithm has linear complexity in the size of the input. Significant research effort has already been spent on parallelizing FA matching [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11]. FA matching has been proven hard to be parallelized and inefficient on parallel architectures, due to an existing dependency between state transitions, i.e., for an FA to perform the next state transition, the result state from the previous transition must be known.

To speed up FA matching on parallel architectures, SFAs have been proposed in [12]. Given an FA A with n states, the corresponding SFA $S(A)$ simulates n parallel instances of FA A . (This simulation is similar to the simulation of a non-deterministic FA by a deterministic FA through the subset construction algorithm [13].) In particular, an SFA state s is a vector of dimension n of FA states; the SFA start-state is the vector $\langle q_0, \dots, q_{n-1} \rangle$, where each q_i is a state of the original FA. An SFA transition from SFA state s_1 to SFA state s_2 on input symbol σ , denoted by $s_1 \xrightarrow{\sigma} s_2$, subsumes the transitions of the underlying FA from each of the FA states in vector s_1 . Running the SFA on a sub-string of the input constitutes n parallel executions of the corresponding FA, each from a unique state from the set of A 's states. Given an SFA, it is then possible to split the input into sub-strings, match each of the sub-strings in parallel with the SFA, and combine the result vectors by reduction.

SFA construction requires considerable time and space in terms of the number of FA states. Therefore, the algorithm mostly becomes intractable for real-world problem sizes. In particular, we found that a large part of the sequence patterns from the PROSITE protein sequence database [14],

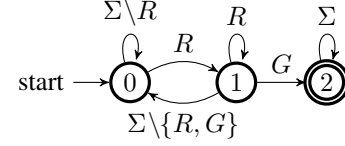
[15] exceeded the computational power of a contemporary 4-CPU (64 physical cores) multicore server with 128 GB of main memory.

The objective of this research is to improve the efficiency of the SFA construction algorithm, to make it tractable for real-world problem sizes. We identify the most time- and space-consuming part of the SFA construction algorithm, which is the representation and comparison of SFA states. We introduce fingerprints to represent SFA state (vectors) by a single 64-bit quantity, which reduces both the program's working set and the time spent for the comparison of SFA states. We introduce a hash-table that hashes SFA states on their fingerprints. This hash-table reduces the set membership test of SFA states to $O(1)$. For our parallelization of the SFA construction algorithm we designed, implemented, and evaluated several approaches to find the most efficient combination of coarse-grained, medium-grained and fine-grained parallelism. Our parallel SFA construction algorithm is non-blocking and we state the termination condition for this algorithm.

This paper makes the following contributions:

- 1) Fingerprints and x86 ISA supporting operations: Performing a membership test on a set of SFA states is an expensive computation due to the increasing number of states during SFA construction. To speed up set membership tests on SFA states, we employ Rabin fingerprints [16], [17]. We adapt the Barrett reduction algorithm [18] and use a special x86 instruction [19], [20] to attain fast fingerprinting.
- 2) Hashing: Determining whether an SFA state has already been generated requires a comparison to all previously generated states. Although the fingerprints suggested in this paper reduce the amount of comparisons, the linear search is still costly. To reduce the number of comparisons to $O(1)$, we introduce hashing that exploits fingerprints as key values.
- 3) Several parallelization approaches: Within the construction algorithm, there are several sources of parallelization. As proposed and presented in former research [21], which targeted a parallel version of the subset construction algorithm, we investigate all possible parallelism sources and determine which benefit SFA construction on multicore platforms. We devise a static work distribution method and optimize all data-structures to increase the locality of memory accesses.
- 4) We evaluate our parallel SFA construction algorithm and SFA-based FA matching for a selection of DNA sequence patterns from the PROSITE protein sequence database [14], [15]. Our evaluation platforms include a 4-CPU (64 physical cores) AMD Opteron system and a 2-CPU (28 cores, 2 hyperthreads per core) Intel Haswell E5-2697 v3 system.

The remainder of this paper is organized as follows. In



(a) Example FA

δ	$\Sigma \setminus \{R, G\}$	R	G
0	0	1	0
1	0	1	2
2	2	2	2

(b) transition table

```

1 state ← 0
2 for i ← 0 to |Str| - 1 do
3   state ← δ(state, Str[i])
  
```

(c) sequential matching routine on input *Str*.

Figure 1: Example FA: state diagram, transition table and matching routine

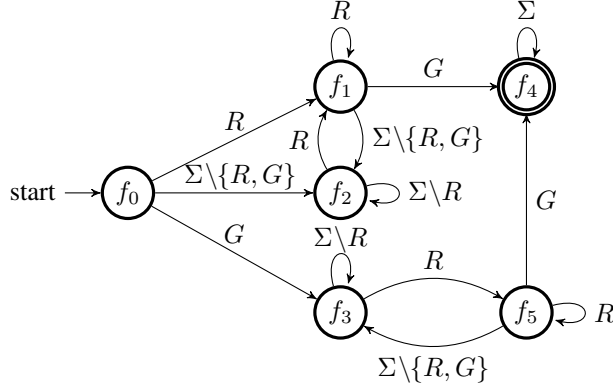
Section II we explain the relevant background material, specifically the sequential SFA construction method and Rabin fingerprints. In Section III we present our optimization methods for the SFA construction algorithm. Section IV contains our experimental evaluation. We discuss the related work in Section V and draw our conclusions in Section VI.

II. BACKGROUND

Finite automata: A tuple $(Q, \Sigma, \delta, I, F)$ describes a deterministic finite automaton (DFA) A . Q is a finite set of states and $|Q|$ is referred to as the size of the DFA. Σ is a finite alphabet of characters and Σ^* is the set of strings over Σ . $I \subseteq Q$ is a set of initial states, but in DFAs there is one initial state $q_0 \in Q$, called the start state. $F \subseteq Q$ is the set of accepting states. δ is a transition function of $Q \times \Sigma \rightarrow Q$. We extend transition function δ to δ^* : $\delta^*(q, ua) = p \Leftrightarrow \delta^*(q, u) = q', \delta(q', a) = p, a \in \Sigma, u \in \Sigma^*$. An input string *Str* over Σ is accepted by DFA A if the DFA contains a labeled path from q_0 to a final state such that this path reads *Str*. The DFA membership test is conducted by computing $\delta^*(q_0, Str)$ and checking whether the result is an accepting state. As a notational convention, we denote the symbol in the i th position of the input string by $Str[i]$.

Fig. 1a shows an example FA over the alphabet of one-letter abbreviations for the 20 amino-acids ($\Sigma = \{A, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W, Y\}$). The FA accepts input strings that contain the sequence *RG* and has start state 0 and a final state 2. The table in Fig. 1b encodes the transition function δ , and Fig. 1c shows a sequential matching routine.

Simultaneous Deterministic Finite Automata: Due to the fundamental reason that the overhead of speculation



(a) Constructed SFA from FA in Fig. 1a

f_0	$0 \rightarrow \{0\}$ $1 \rightarrow \{1\}$ $2 \rightarrow \{2\}$	f_3	$0 \rightarrow \{0\}$ $1 \rightarrow \{2\}$ $2 \rightarrow \{2\}$
f_1	$0 \rightarrow \{1\}$ $1 \rightarrow \{1\}$ $2 \rightarrow \{2\}$	f_4	$0 \rightarrow \{2\}$ $1 \rightarrow \{2\}$ $2 \rightarrow \{2\}$
f_2	$0 \rightarrow \{0\}$ $1 \rightarrow \{0\}$ $2 \rightarrow \{2\}$	f_5	$0 \rightarrow \{1\}$ $1 \rightarrow \{2\}$ $2 \rightarrow \{2\}$

(b) State mapping table

Figure 2: Example SFA: state diagram and state mapping table

Algorithm 1: SFA Construction

Require: Automaton $\mathcal{A} = (Q, \Sigma, \delta, I, F)$

Ensure : SFA $\mathcal{S} = (Q_s, \Sigma, \delta_s, I_s, F_s)$ is equivalent to an automaton \mathcal{A}

```

1  $Q_s \leftarrow \emptyset, Q_{tmp} \leftarrow \{f_I\}$ 
2 while  $Q_{tmp} \neq \emptyset$  do
3   choose and remove a mapping  $f$  from  $Q_{tmp}$ 
4    $Q_s \leftarrow Q_s \cup \{f\}$ 
5   forall the  $\sigma \in \Sigma$  do
6      $q \in Q$   $f_{next}(q) := \bigcup_{q' \in f(q)} \delta(q', \sigma)$ 
7      $\delta_s[f, \sigma] \leftarrow f_{next}$ 
8     if  $f_{next} \notin Q_s, Q_{tmp}$  then
9        $Q_{tmp} \leftarrow Q_{tmp} \cup \{f_{next}\}$ 
10  $I_s \leftarrow \{f_I\}$ 
11  $F_s \leftarrow \{f \in Q_s \mid \exists q \in I \mid f(q) \cap F \neq \emptyset\}$ 

```

is inevitable for parallel FA matching, SFA is introduced [12]. SFA is a model of efficient parallelization for FA matching, generated by the original FA. Algorithm 1 denotes the construction algorithm to develop SFAs. Details of the algorithm can be found in [12].

Fig. 2a is an example SFA and Fig. 2b is the corresponding state mapping table, generated by the FA in Fig. 1a.

In summary, the construction algorithm keeps finding new SFA-states, checking whether they are already in a set and adding them if they are not in the set yet, starting with the set that has only the start state f_I . After the algorithm is finished, a SFA and a corresponding state mapping table are finally produced. For the example SFA, Q_s , whose elements in Fig. 2a are $f_i, 0 \leq i \leq 5$.

Rabin Fingerprints: Fingerprints are short bit-strings for larger objects. If two fingerprints are different, the corresponding objects are known to be different. There is a small probability that two objects map to the same fingerprint, which is called a collision. Michael O. Rabin's fingerprinting method [17], [16] creates fingerprints from arbitrary bit-strings $A = \{a_1, a_2, \dots, a_m\}$ by interpreting A as a polynomial $A(t)$ of degree $m - 1$ with coefficients in \mathbb{Z}_2 .

$$A(t) = a_1 t^{m-1} + a_2 t^{m-2} + \dots + a_m. \quad (1)$$

To create a k -bit fingerprint, an irreducible random polynomial $\mathcal{P}(t)$ over \mathbb{Z}_2 of degree k is selected. The Rabin fingerprint can then be defined as

$$f(A) = A(t) \bmod \mathcal{P}(t). \quad (2)$$

As shown in [16], the probability of a collision among a set of n distinct bit-strings is less than $\frac{n^2 m}{2^k}$.

To apply Rabin fingerprints, the costs for computing the polynomial modulo operation from Eq. (2) must be kept to a minimum. The Barrett reduction method from [18] can be used to express computationally costly modulo operations in terms of multiplication, division, addition and the floor function. For arbitrary integers a and n , the basic idea of Barrett reduction is denoted as

$$a \bmod n = a - \lfloor am \rfloor n, \text{ where } m = 1/n. \quad (3)$$

Rabin's fingerprinting method $A(t) \bmod \mathcal{P}(t)$ can be calculated through Barrett reduction as

$$A(t) \bmod \mathcal{P}(t) = A(t) \oplus \left[\left(\lfloor A(t)/t^k \rfloor \bullet \lfloor t^{2k}/\mathcal{P}(t) \rfloor \right) \div t^k \right] \bullet \mathcal{P}(t), \quad (4)$$

where \oplus and \bullet denote addition and multiplication in the Galois Fields of characteristic 2 ($\text{GF}(2^k)$). In particular, \bullet is a carry-less multiplication operation.

$$\begin{aligned}
T1_{pre} &= \lfloor A(t)/t^k \rfloor, \quad M = \lfloor t^{2k}/\mathcal{P}(t) \rfloor \\
T1 &= T1_{pre} \bullet M \\
T2_{pre} &= \lfloor T1 \div t^k \rfloor \\
T2 &= T2_{pre} \bullet \mathcal{P}(t) \\
A(t) \bmod \mathcal{P}(t) &= A(t) \oplus T2
\end{aligned} \quad (5)$$

Eq. (5) displays the work-flow of the fingerprinting algorithm from Eq. (4) step by step.

The x86 architecture provides the `PCLMULQDQ` SSE instruction to perform carry-less multiplication efficiently in hardware (according to [22], the savings of the `PCLMULQDQ` instruction are 100 cycles per multiplication).

The `PCLMULQDQ` instruction takes two 64 bit operands and returns a 128 bit result. The SFA states we will fingerprint are mostly larger than 64 bit. Even if we store an FA state as type `unsigned short` (16 bit), only 4 states fit within 64 bit. We apply the folding method described in [19] to reduce SFA states of arbitrary size to 64 bit fingerprints.

III. OPTIMIZING THE SFA CONSTRUCTION ALGORITHM

$$\mathcal{O}\left(\sum_{i=1}^{|Q_s|} \sum_{j=1}^{|\Sigma|} (|Q| + |Q| \times i)\right) = \mathcal{O}\left(\frac{1}{2} \times |\Sigma| \times |Q| \times |Q_s| \times (|Q_s| + 3)\right). \quad (6)$$

The time complexity of Algorithm 1 can be determined as follows; because the outmost while loop (line 2) keeps working until no more elements are left in Q_{tmp} , while loop iterates $|Q_s|$ time, one per SFA state, same as the size of the resulting SFA. Inside of while loop, next states for each SFA state are calculated on each symbol one by one in for loop (line 5). Thus for loop iterates $|\Sigma|$ times per SFA state. To decide whether a newly created state is already in the set Q_s , the “exhaustive” set membership test (comparing all FA states in SFA state) between the new state and all other states in sets should be done. If we presume in the worst-case that all states are different, set membership test should be done as same as the number of states in the set at that moment and each membership test need $|Q|$ comparisons of FA state. Therefore, the time complexity of this algorithm on worst-case is given in the equation (6).

From the time complexity, we can recognize that the number of generated SFA states is the most significant factor for time consumption of this algorithm. As we can see, comparisons for newly generated states take the most of the time $\mathcal{O}(|\Sigma| \times |Q| \times |Q_s| \times (|Q_s| + 1) \times \frac{1}{2})$. The part for checking termination $\mathcal{O}(|Q_s|)$ and state transition $\mathcal{O}(|Q_s| \times |\Sigma| \times |Q|)$ only take a very small portion of the whole algorithm. This means we need to focus on decreasing time taken by the state comparison part in the following optimization.

A. Hashing with Fingerprints

If we adapt fingerprints, then all states already have fingerprints and they will just compare the value of fingerprints with each other. The exhaustive matching is only conducted when two fingerprints from states are identical. However, without fingerprints, states will compare their own contents with each other by exhaustive comparison. As a result, our algorithm is the non-probabilistic approach which always constructs the exact SFA.

By exploiting fingerprints further, we introduce a hash table data structure to our construction algorithm. This can be done easily because fingerprints are stored in an `unsigned int` or `unsigned long long` data type (`uint32_t` or `uint64_t`) so that we can use it directly as a key value for hashing.

We declare the hash table as a pointer array to indicate each SFA state node. During construction algorithm, a fingerprint for each generated state is calculated before the set membership test. Each hash table entry uses this information to point out a SFA state node which has corresponding fingerprint value.

If there is no duplication in fingerprints, the generated state is added to the hash table at the entry of hash index. Otherwise, the new state should follow the pointer chain from its hash entry. While going to the leaf of linked list pointer chain, comparison with states in the chain needed to be done. If the new state cannot meet the same state until it reaches the leaf node, it is added to the leaf of the linked list chain. At the best case, now we only need $\mathcal{O}(1)$ time to find states that have the same fingerprint.

B. Parallelization for Multicore Architectures

1) *Sources of Parallelism:* As presented in the work of parallelizing the subset construction algorithm [21], the sequential SFA construction algorithm presented in Algorithm 1 also has similar parallel sources in itself, from coarse-grained parallelism to fine-grained parallelism.

The fine-grained parallel source is not practical. We could divide state transitions (line 6) of FA states but total work amount for state transition is rather small ($|Q_s| \times |Q|$) compared to the state comparison part to gain some recognizable benefit. Also, it's questionable how we can parallelize it smart enough when the number of participating processor exceeds the number of FA states. It is hard to distribute and assign work to processors without making idle processors. Moreover, after splitting the work, the partial result of a transition should be gathered to make a new SFA state. It means introduction of synchronization and waiting overhead between threads. We might gain some advantage from the cache locality because transitions are split based on the FA states and one row in the transition table δ corresponds to the transition information of one FA state. But still, compared what we could gain from the fine-grained parallelization to the overhead of synchronization, this approach is hard to be a good option for task-level parallelization.

The most significant and easy to recognize source is coarse-grained parallelism which is dividing the work based on the SFA states. In this case, the while loop (from line 2 to 9) is parallelized. However, this parallelization can be meaningful only if unprocessed states in the Q_{tmp} are always in a plenty number so that all processors can have a state to process in most of time during construction and not

to go into idle state. This approach would be appropriate for FAs which have a large number of states.

Medium-grained parallelism is dividing tasks based on symbols. Each thread can do series of works by itself, without help of other threads; producing new states on given symbols, testing the possible duplication of states and adding them to the set. Now synchronization is not needed because each thread can proceed individually. However, still we need a good scheme to distribute work to processors so that every processor has the same amount of work. Also, consideration of when the number of processors is greater than the number of symbols should be needed.

We utilize coarse-grained parallelism for the transposed transition table and medium-grained parallelism for static work allocation.

Algorithm 2: Parallel SFA Construction with static distribution (fewer threads than symbols)

Require: Automaton $\mathcal{A} = (Q, \Sigma, \delta, I, F)$, $|P|$ = the number of threads,
Distribution of Σ , $D = \{B_0, \dots, B_{P-1}\}$,
current threads: j th thread

Ensure : SFA $\mathcal{S} = (Q_s, \Sigma, \delta_s, I_s, F_s)$ is equivalent to an automaton \mathcal{A}

```

1  $Q_s \leftarrow \{f_I\}, Index_j \leftarrow 0$ 
2 while  $\exists Q_s[Index_j] \neq \emptyset$  do in parallel
3   while  $\forall \sigma_i \in B_j$  do
4     choose a mapping  $f$  from  $Q_s[Index_j]$ 
5      $q \in Q \quad f_{next}(q) := \bigcup_{q' \in f(q)} \delta(q', \sigma_i)$ 
6      $\delta_s[f, \sigma_i] \leftarrow f_{next}$ 
7     if  $f_{next} \notin Q_s$  then
8        $Q_s \leftarrow Q_s \cup \{f_{next}\}$ 
9      $Index_j \leftarrow Index_j + 1$ 
10  $I_s \leftarrow \{f_I\}$ 
11  $F_s \leftarrow \{f \in q_s | \exists q \in I | f(q) \cap F \neq \emptyset\}$ 

```

2) *Static Work Allocation:* Algorithm 2 describes the parallel algorithm with the static distribution of symbols when the number of threads $|P|$ is less than the number of symbols $|\Sigma|$. An array that has distribution information for each thread is introduced. Because the number of threads is less than the number of symbols, every thread has to compute one or more symbols and we need a variable to hold those data. A vector $D = \{B_0, \dots, B_{P-1}\}$ has the distribution information of symbols B_j about what symbols are assigned to the thread j . Thus, iteration for symbols (line 3) is limited to only assigned symbols in B_j . The inside of iteration, lines from 4 to 9, each thread computes next nodes with assigned symbols. When newly created state f_{next} is not in the set Q_s , it is added to Q_s . This algorithm finishes when all threads do not have any non-processed SFA state (line 2).

Algorithm 3: Parallel SFA Construction with static distribution by grouping

Require: Automaton $\mathcal{A} = (Q, \Sigma, \delta, I, F)$, g = the number of groups,
Partition of Q_s , $D = \{S_0, \dots, S_{g-1}\}$,
threads: j th thread in k th group

Ensure : SFA $\mathcal{S} = (Q_s, \Sigma, \delta_s, I_s, F_s)$ is equivalent to an automaton \mathcal{A}

```

1  $Q_s \leftarrow \{f_I\}, Index_{k*g+j} \leftarrow 0$ 
2 while  $\exists Q_s[Index_{k*g+j}] \neq \emptyset$  do in parallel
3   while  $Q_s[Index_{k*g+j}] \in S_k$  do
4     choose a mapping  $f$  from  $Q_s[Index_{k*g+j}]$ 
5      $q \in Q \quad f_{next}(q) := \bigcup_{q' \in f(q)} \delta(q', \sigma_j)$ 
6      $\delta_s[f, \sigma_j] \leftarrow f_{next}$ 
7     if  $f_{next} \notin Q_s$  then
8        $Q_s \leftarrow Q_s \cup \{f_{next}\}$ 
9      $Index_{k*g+j} \leftarrow Index_{k*g+j} + 1$ 
10  $I_s \leftarrow \{f_I\}$ 
11  $F_s \leftarrow \{f \in q_s | \exists q \in I | f(q) \cap F \neq \emptyset\}$ 

```

We also introduce an algorithm for the opposite case when the number of threads is greater than the number of symbols. In this case, we can consider two cases, whether the number of threads is a multiple of the number of symbols or not. The following Algorithm 3 explains the former case. In this case, because the number of threads is a multiple of the number of symbols, $|\Sigma|$ threads are gathered together to form a group. By doing this, each thread in a group will only take one symbol to process. Thus, SFA states set Q_{tmp} should be split into S_0, \dots, S_{g-1} and assigned to each group, where g is the number of groups.

Now an array D holds distribution information of SFA states for each group from S_0 to S_{g-1} when the number of groups is g . Threads in the group k will only choose a SFA state in the distribution set S_k . Also because the number of threads in a group is the same as the number of symbols $|\Sigma|$, the j th thread in a group will only compute next states on symbol σ_j . Please note that there is no for loop that iterates over symbols, unlike in the previous algorithm. In other words, each group just cares about SFA states assigned to them and each thread also computes the only symbol given to them. Thus, a thread in the k th group starts its execution if there is a state in S_k that is not processed for the symbol which is given to it (line 3). The algorithm between line 4 to 9 is the same as in the previous algorithm.

For the last condition, when the number of threads is not a multiple of but greater than the number of symbols, the parallel algorithm with static work distribution is a mixture of the above two Algorithms 2 and 3. In this case, the number of groups g will be $\lceil \frac{|P|}{|\Sigma|} \rceil$ where $|P|$ is the number of threads and $|\Sigma|$ is the number of symbols. Among them,

groups of full threads which have $|\Sigma|$ threads will act as presented in Algorithm 3. (The number of such groups will be $\lceil \frac{|p|}{|\Sigma|} \rceil - \lfloor \frac{|p|}{|\Sigma|} \rfloor$.) The remaining groups which do not have enough threads to have one symbol per thread, will follow Algorithm 2 except for line 2, because they also choose a state f in the set S_{g-1} , which is the partition of Q_{tmp} for the last group. Thus, threads in the last group follow Algorithm 3 only for line 2. Therefore, especially in this case, the distribution of SFA states in Q_s should be done by considering the number of threads for each group.

3) *Transposing Transition Tables*: Transition tables of FAs are represented as 2-dimensional arrays in row-major layout. With those transition tables, each thread has a high probability to access different rows of them whenever it computes the next FA state according to the current FA state and symbol. This decreases the locality of memory accesses and eventually aggravates performance of SFA construction.

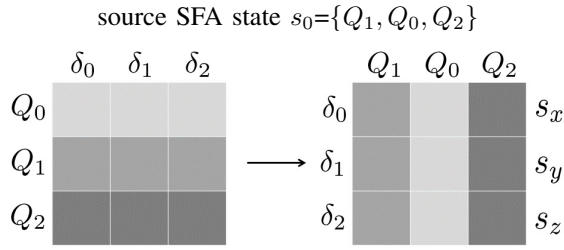


Figure 3: Example transposed transition table

As an example, suppose our algorithm is going to derive new SFA states $\{s_x, s_y, s_z\}$ from a given SFA state s_0 on a set of input symbols $\{\delta_0, \delta_1, \delta_2\}$. We transpose the transition table δ according to the DFA states in s_0 . As follows from the left part of Fig. 3, the transition table is read in row-major order, with each row corresponding to one SFA state in s_0 . Each such row becomes one column in the transposed table in the right part of Fig. 3. Each *row* in the transposed table represents a new SFA state (which will again be processed in row-major order, to facilitate locality).

The sequential algorithm of this approach is the same as Algorithm 1 except for line 6 is replaced by transposing. Because transposing a transition table produces next SFA states according to all symbols, coarse-grained parallelism is appropriate for this approach. Therefore, each thread takes one SFA state from the list of non-computed states until every thread has no SFA states to process.

IV. EXPERIMENTAL RESULTS

In this section, we demonstrate how all optimization methods presented in Section III affect the performance of SFA construction algorithm. We implemented SFA construction and matching for two architectures summarized in Table I. POSIX threads [23] were used to parallelize SFA construction and matching across multiple cores. To generate minimal DFAs from regular expressions, we use Grail+ [24],

Name	CPU Model	CPUs	$\frac{\text{Cores(HTs)}}{\text{CPU}}$	Clock Freq.
Intel Xeon	Intel Core E5-2697 v3	2	14(28)	2.60 GHz
AMD Opteron	AMD Opteron Processor 6378	4	16	2.40 GHz

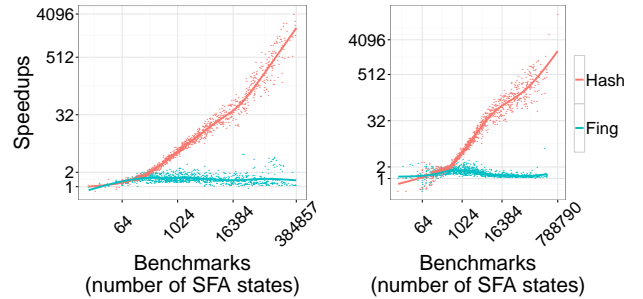
Table I: Hardware Specifications

[25]. Our SFA construction and matching frameworks read DFAs and input strings in Grail+ format and convert them to our framework's internal representation.

From the smallest benchmark with 5 DFA states to the largest benchmark with 2930 DFA states, total 1062 DFAs from PROSITE protein database [14] are used.

The interpolated lines of diagrams were created using R's local regression method.

A. Optimizations for Sequential Algorithm



(a) on the AMD system

(b) on the Intel system

Figure 4: Performance of fingerprints and hashing

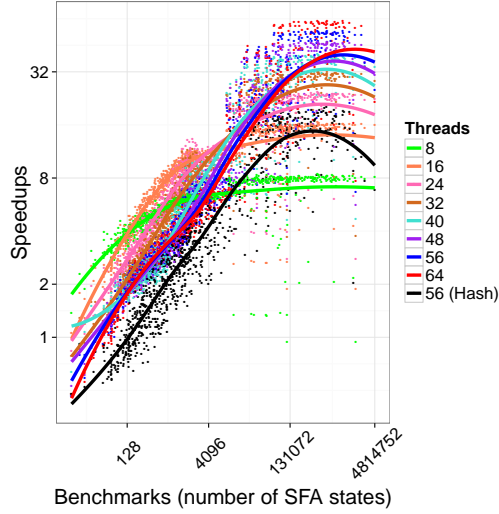
Figure 4 illustrates the speedup of fingerprints compared to the non-optimized baseline and the speedup of hashing compared to fingerprints. Therefore, the total speedup is the multiplication of both of them.

Because of time limit, it is impossible to experiment all benchmarks for the sequential methods, especially for the baseline. Without fingerprints and hashing, the execution time for one benchmark might take over one day. And we show the speedups of benchmarks which are tractable on our servers. This explains the reason why we need to parallelize the sequential SFA construction.

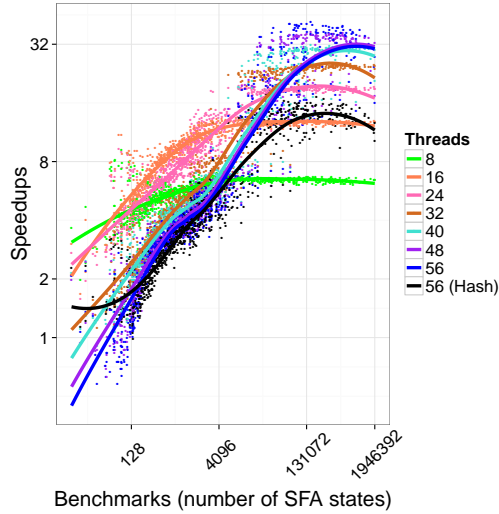
B. Parallelization

Figure 5 describes the speedup of our parallel SFA construction algorithm with fingerprints, hashing and transposing over the best optimized sequential implementation. Thus, the overall speedup compared to the baseline sequential algorithm can be achieved by multiplying the parallel speedup, the sequential hashing speedup and the sequential fingerprints speedup.

Because the Intel system has only half (64 GB) of the memory size of the AMD system, the range of benchmarks



(a) on the AMD system



(b) on the Intel system

Figure 5: Speedup of the best parallel implementation over the best sequential implementation.

we could run are different depending on the system. The maximum speedups achieved by parallelization on the AMD and the Intel system are 62.365x with 64 threads and 41.438x with 56 threads while the median speedups of them are 3.968x and 3.911x respectively. The reason why median speedups are different from the maximum speedups is that there is not enough work to distribute in the case of small benchmarks which generate smaller numbers of SFA states. In that situation, a smaller number of threads tends to show better performance, because more threads incur a penalty from inter-core cache coherence transfers (larger SFA sizes do not fit into the CPU caches anymore and rely more on data from main memory).

C. SFA Matching

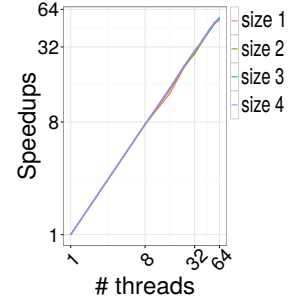


Figure 6: SFA matching

The result of SFA matching with an input string of ten billion characters is shown in Figure 6. The experiment has been conducted on the 64-core AMD Opteron system and we picked the median execution time of three iterations for each benchmark. The numbers of SFA states are 12299, 25332, 206351 and 2437146 respectively (size 1, size 2, size 3 and size 4 in Figure 6). Because there is no dependency between the input chunks, the speedup of SFA matching over sequential matching is linear in the number of threads. It is worth noting that this speedup has been observed even with the large, 2,437,146-state SFA. The transition table of this SFA has a size of 97 MB, which exceeds the 12 MB L3 cache of each of the 4 Opteron 6378 CPUs already considerably. We thus expect that SFA matching will scale to even larger SFAs.

V. RELATED WORK

To the best of our knowledge, there are no challenges for parallelizing the SFA construction so far. In this section, we only can introduce research efforts that do parallelization of FA matching, which is based on speculation.

Locating a string in a larger text has applications with text editing, compiler front-ends and web browsers, internet search engines, computer security, and DNA sequence analysis. Early string searching algorithms such as Aho–Corasick [26], Boyer–Moore [27] and Rabin–Karp [28] efficiently match a finite set of input strings against an input text.

Regular expressions allow the specification of infinite sets of input strings. Converting a regular expression to a DFA for DFA membership tests is a standard technique to perform regular expression matching. The specification of virus signatures in intrusion prevention systems [29], [30], [31] and the specification of DNA sequences [32], [33] constitute recent applications of regular expression matching with DFAs.

Considerable research effort has been spent on parallel algorithms for DFA membership tests. Ladner et al. [1] applied the parallel prefix computation for DFA membership tests with Mealy machines. Hillis and Steele [2] applied

parallel prefix computations for DFA membership tests on the 65,536 processor Connection Machine. Ravikumar’s survey [34] shows how DFA membership tests can be stated as a chained product of matrices. Because of the underlying parallel prefix computation, all three approaches perform a DFA membership test on input size n in $\mathcal{O}(\log(n))$ steps, requiring n processors. Their algorithms handle arbitrary regular expressions, but the underlying assumption of a massive number of available processors can hardly be met in most practical settings. Misra [3] derived another $\mathcal{O}(\log(n))$ string matching algorithm. The number of required processors is on the order of the product of the two string lengths and hence not practical.

A straight-forward way to exploit parallelism with DFA membership tests is to run a single DFA on multiple input streams in parallel, or to run multiple DFAs in parallel. This approach has been taken by Scarpazza et al. [4] with a DFA-based string matching system for network security on the IBM Cell BE processor. Similarly, Wang et al. [8] investigated parallel architectures for packet inspection based on DFAs. Both approaches assume multiple input streams and a vast number of patterns (i.e., virus signatures), which is common with network security applications. However, neither approach parallelizes the DFA membership algorithm itself, which is required to improve applications with single, long-running membership tests such as DNA sequence analysis.

Scarpazza et al. utilize the SIMD units of the Cell BE’s synergistic processing units to match multiple input streams in parallel. However, their vectorized DFA matching algorithm contains several SISD instructions and the reported speedup from 16-way vectorization is only a factor of 2.51.

Recent research efforts focused on speculative computations to parallelize DFA membership tests. Holub and Štekr [5] were the first to split the input string into chunks and distribute chunks among available processors. Their speculation introduces a substantial amount of redundant computation, which restricts the obtainable speedup for general DFAs to $\mathcal{O}(\frac{|P|}{|Q|})$, where $|P|$ is the number of processors, and $|Q|$ is the number of DFA states. Their algorithm degenerates to a speed-down when $|q|$ exceeds the number of processors. To overcome this problem, Holub and Štekr specialized their algorithm for k -local DFAs. A DFA is k -local if for every word of length k and for all states $p, q \in Q$ it holds that $\delta^*(p, w) = \delta^*(q, w)$. Starting the matching operation k symbols ahead of a given chunk will synchronize the DFA into the correct initial state by the time matching reaches the beginning of the chunk, which eliminates all speculative computation. Holub and Štekr achieve a linear speedup of $\mathcal{O}(|P|)$ for k -local automata.

Jones et al. [6] reported that with the IE 8 and Firefox web browsers 3–40% of the execution-time is spent parsing HTML documents. To speed up browsing, Jones et al. employ speculation to parallelize token detection (lexing) of HTML language front-ends. Similar to Holub and Štekr’s

k -local automata, they use the preceding k characters of a chunk to synchronize a DFA to a particular state. Unlike k -locality, which is a static DFA property, Jones et al. speculate the DFA to be in a particular, frequently occurring DFA state at the beginning of a chunk. Speculation fails if the DFA turns out to be in a different state, in which case the chunk needs to be re-matched. Lexing HTML documents results in frequent matches, and the structure of regular expressions is reported to be simpler than, e.g., virus signatures [9]. Speculation is facilitated by the fact that the state at the beginning of a token is always the same, regardless where lexing started. A prototype implementation is reported to scale up to six of the eight synergistic processing units of the Cell BE.

The speculative parallel pattern matching (SPPM) approach by Luchaup et al. [9], [7] uses speculation to match the increasing network line-speeds faced by intrusion prevention systems. SPPM DFAs represent virus signatures. Like Jones et al., DFAs are speculated to be in a particular, frequently occurring DFA state at the beginning of a chunk. SPPM starts the speculative matching at the beginning of each chunk. With every input character, a speculative matching process stores the encountered DFA state for subsequent reference. Speculation fails if the DFA turns out to be in a different state at the beginning of a speculatively matched chunk. In this case, re-matching continues until the DFA synchronizes with the saved history state (in the worst case, the whole chunk needs to be re-matched). A single-threaded SPPM version is proposed to improve performance by issuing multiple independent memory accesses in parallel. Such pipelining (or interleaving) of DFA matches is orthogonal to our approach, which focuses on latency rather than throughput.

SPPM assumes all regular expressions to be suffix-closed, which is the common scenario with intrusion prevention systems; A regular expression is suffix-closed if matching a given string w implies that w followed by any suffix is matched, too. A suffix-closed regular language has the property that $x \in L \Leftrightarrow \forall w \in \Sigma^* : xw \in L$.

The speculative parallel DFA membership test reported by Ko et al. [10] is to parallelize DFA membership tests for multicore, SIMD, and cloud computing environments. This technique is one of the speculative parallel matching methods by searching arbitrary regular expressions. It requires dividing the input string into chunks, matching chunks in parallel, and combining the matching results. When the input string is partitioned, the algorithm decides the amount of characters of each chunk depending on the number of possible start states. Unlike the previous approaches of parallel membership tests, it is *failure-free*, which means speed-downs never happen by maintaining the sequential semantics.

Another approach of parallel pattern matching was investigated by Mytkowicz et al. [11]. It provides a parallel

algorithm for DFAs, which breaks the dependencies of state transitions by enumerating computations from all possible start states on each input character. With the enumeration algorithm, the number of state transitions decreases even though all states are considered as possible start states at the beginning of matching.

To remove the overhead from the speculative parallel pattern matching, a simultaneous finite automaton is introduced by Sin'ya et al. [12]. Because the main problem of parallel pattern matching is the dependency of state transitions, it extends an automaton so that it involves the simulation of state transitions. Although it can remove the dependency, extending an automaton to a SFA is costly because the algorithm is based on the well-known subset construction algorithm, which requires considerable time. And our approach makes their algorithm efficiently by parallelizing it.

One of our optimization methods is hashing and we utilized hashing to accomplish faster execution. Stern et al. [35] applied hashing to save memory because the verification of complex protocols suffered from the state explosion problem. Even though we have the same state explosion problem, we approached it in the non-probabilistic way whereas they considered the probabilistic method by compacting states in the hash table.

VI. CONCLUSIONS

Although string pattern matching with FAs has been widely used in various areas, parallelizing its sequential algorithm was difficult because of the dependency of state transitions. As a solution to break this, SFAs were introduced. Although SFA matching achieved a lot of speedups, constructing SFAs was costly in terms of execution time because the algorithm was implemented sequentially and required many number of computations. Also, the size of an SFA is significantly bigger than that of an original DFA.

In this paper, we presented several optimization methods for the SFA construction algorithm on multicore architecture platforms. We adapted fingerprints method provided by Rabin with the help of Barrett reduction algorithm and AES, SSE instruction set to reduce the number of comparisons of SFA states because a comparison requires considerable time. Hashing was introduced to support fingerprints for further optimizations and it derived the $\mathcal{O}(1)$ time complexity of the algorithm in most cases. Our proposed algorithm is nonblocking and exploits various parallelism sources after investigation; instruction-level, data-level, and task-level parallelism with coarse, medium, and fine granularities. Thread level parallelization is used for each parallelism sources; instruction, data and task-level parallelism. We also suggested the static distribution calculates appropriate amount for each thread in advance so every thread can have and process almost same amount of work simultaneously. And transposing the transition table was implemented in consideration of the locality of memory accesses. As a result,

our overall speedups over the sequential baseline which we could observe on our servers are up to 118541x on the AMD system and 2113968x on the Intel system.

REFERENCES

- [1] R. E. Ladner and M. J. Fischer, "Parallel prefix computation," *J. ACM*, vol. 27, no. 4, pp. 831–838, Oct. 1980.
- [2] W. D. Hillis and G. L. Steele, Jr., "Data parallel algorithms," *Commun. ACM*, vol. 29, no. 12, pp. 1170–1183, Dec. 1986.
- [3] J. Misra, "Derivation of a parallel string matching algorithm," *Inf. Process. Lett.*, vol. 85, no. 5, pp. 255–260, Mar. 2003.
- [4] D. P. Scarpazza, O. Villa, and F. Petrini, "Peak-performance DFA-based string matching on the Cell processor," in *21th International Parallel and Distributed Processing Symposium*, 2007, pp. 1–8.
- [5] J. Holub and S. Štekr, "On parallel implementations of deterministic finite automata," in *Proceedings of the 14th International Conference on Implementation and Application of Automata*, 2009, pp. 54–64.
- [6] C. G. Jones, R. Liu, L. Meyerovich, K. Asanović, and R. Bodík, "Parallelizing the web browser," in *Proceedings of the First USENIX conference on Hot topics in parallelism*, ser. HotPar'09. Berkeley, CA, USA: USENIX Association, 2009, pp. 7–7.
- [7] D. Luchaup, R. Smith, C. Estan, and S. Jha, "Multi-byte regular expression matching with speculation," in *Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection*. Springer, 2009, pp. 284–303.
- [8] X. Wang, K. He, and B. Liu, "Parallel architecture for high throughput DFA-based deep packet inspection," in *2010 IEEE International Conference on Communications*, 2010, pp. 1–5.
- [9] D. Luchaup, R. Smith, C. Estan, and S. Jha, "Speculative parallel pattern matching," *IEEE Transactions on Information Forensics and Security*, vol. 6, no. 2, pp. 438–451, 2011.
- [10] Y. Ko, M. Jung, Y. Han, and B. Burgstaller, "A speculative parallel DFA membership test for multicore, SIMD and cloud computing environments," *International Journal of Parallel Programming*, vol. 42, no. 3, pp. 456–489, 2014.
- [11] T. Mytkowicz, M. Musuvathi, and W. Schulte, "Data-parallel finite-state machines." Architectural Support for Programming Languages and Operating Systems (ASPLOS), March 2014.
- [12] R. Sin'ya, K. Matsuzaki, and M. Sassa, "Simultaneous finite automata: An efficient data-parallel model for regular expression matching," in *42nd International Conference on Parallel Processing (ICPP)*, Oct 2013, pp. 220–229.
- [13] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, 3rd ed. Addison-Wesley, 2006.
- [14] PROSITE Web Site, "http://prosite.expasy.org," retrieved Oct. 2015.

- [15] A. Gattiker, E. Gasteiger, and A. Bairoch, "ScanProsite: a reference implementation of a PROSITE scanning tool," *Applied Bioinformatics*, vol. 1, no. 2, 2002.
- [16] A. Z. Broder, "Some applications of Rabin's fingerprinting method," in *Sequences II: Methods in Communications, Security, and Computer Science*. Springer, 1993, pp. 143–152.
- [17] M. O. Rabin, "Fingerprinting by random polynomials," Center of Research in Computer Technology, Harvard University, Tech. Rep. TR-15-81, 1981.
- [18] P. Barrett, "Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor," in *Advances in Cryptology CRYPTO 86*, ser. LNCS, A. M. Odlyzko, Ed., vol. 263. Springer, 1987, pp. 311–323.
- [19] V. Gopal, E. Ozturk, J. Guilford, G. Wolrich, W. Feghali, M. Dixon, and D. Karakoyunlu, "Fast CRC computation for generic polynomials using PCLMULQDQ instruction," Intel, Tech. Rep. (white paper), December 2009.
- [20] Intel Advanced Vector Extensions Programming Reference, "<http://software.intel.com/en-us/avx>," retrieved Oct. 2015, June 2011 version.
- [21] H. Choi and B. Burgstaller, "Non-blocking parallel subset construction on shared-memory multicore architectures," in *Proceedings of the Eleventh Australasian Symposium on Parallel and Distributed Computing-Volume 140*. Australian Computer Society, Inc., 2013, pp. 13–20.
- [22] S. Gueron and M. E. Kounavis, "Intel carry-less multiplication instruction and its usage for computing the GCM mode," Intel, Tech. Rep. (white paper), April 2014.
- [23] D. R. Butenhof, *Programming with POSIX threads*. Boston, MA, USA: Addison-Wesley, 1997.
- [24] D. Raymond and D. Wood, "Grail: A C++ library for automata and expressions," *Journal of Symbolic Computation*, vol. 17, pp. 17–341, 1995.
- [25] Grail+ Project Web Site, "<http://www.csd.uwo.ca/Research/grail/>," retrieved Oct. 2015.
- [26] A. V. Aho and M. J. Corasick, "Efficient string matching: an aid to bibliographic search," *Commun. ACM*, vol. 18, no. 6, pp. 333–340, 1975.
- [27] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," *Commun. ACM*, vol. 20, no. 10, pp. 762–772, 1977.
- [28] R. M. Karp and M. O. Rabin, "Efficient randomized pattern-matching algorithms," *IBM J. Res. Dev.*, vol. 31, no. 2, pp. 249–260, 1987.
- [29] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha, "Towards automatic generation of vulnerability-based signatures," in *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, ser. SP '06. IEEE Computer Society, 2006, pp. 2–16.
- [30] R. Sommer and V. Paxson, "Enhancing byte-level network intrusion detection signatures with context," in *Proceedings of the 10th ACM conference on Computer and communications security*, ser. CCS '03. ACM, 2003, pp. 262–271.
- [31] M. Roesch, "Snort—Lightweight Intrusion Detection for Networks," in *Proceedings of the 13th USENIX conference on System administration*, ser. LISA '99. USENIX Association, 1999, pp. 229–238.
- [32] C. Sigrist, L. Cerutti, E. De Castro, P. Langendijk-Genevaux, V. Bulliard, A. Bairoch, and N. Hulo, "PROSITE, a protein domain database for functional characterization and annotation," *Nucleic acids research*, vol. 38, p. D161, 2010.
- [33] B. Boeckmann, A. Bairoch, R. Apweiler, M. Blatter, A. Estreicher, E. Gasteiger, M. Martin, K. Michoud, C. O'Donovan, I. Phan *et al.*, "The SWISS-PROT protein knowledgebase," *Nucleic acids research*, vol. 31, no. 1, p. 365, 2003.
- [34] B. Ravikumar, "Parallel algorithms for finite automata problems," in *IPPS/SPDP Workshops*, vol. 1388. Springer, 1998.
- [35] U. Stern and D. L. Dill, "Improved probabilistic verification by hash compaction," in *In Advanced Research Working Conference on Correct Hardware Design and Verification Methods*. Springer-Verlag, 1995, pp. 206–224.